April 2022,
IPT Course Java

Design Patterns. IO. NIO. NIO2

Trayan Iliev

tiliev@iproduct.org

http://iproduct.org

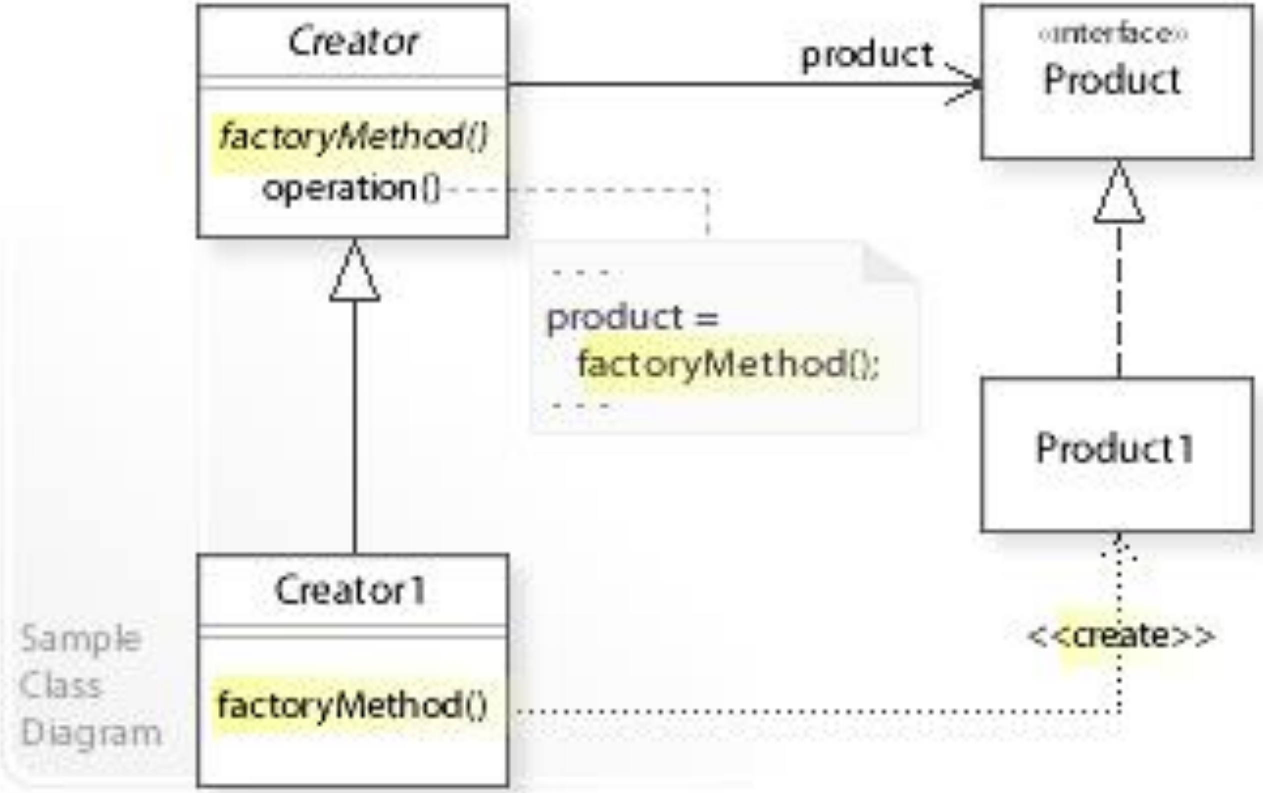
SOLID Design Principles of OOP

- Single responsibility principle - a class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.
- Open–closed principle - software entities should be open for extension, but closed for modification.
- Liskov substitution principle - Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- Interface segregation principle - Many client-specific interfaces are better than one general-purpose interface.
- Dependency inversion principle - depend upon abstractions, not concretions.

Factory Method Design Pattern

- **Factory method** pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created.
- This is done by creating objects by calling a factory method – either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes – rather than by calling a constructor.

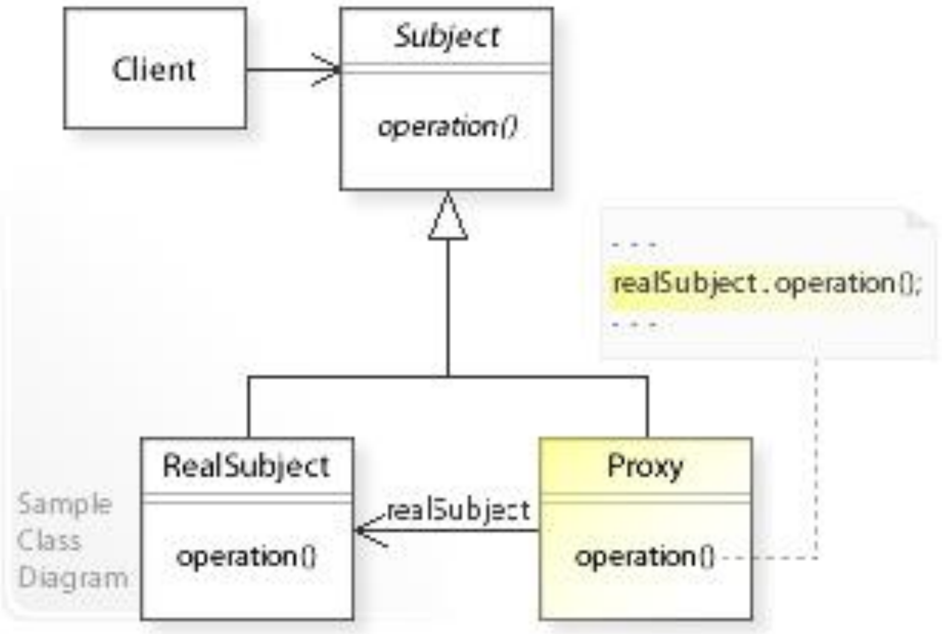
Factory Method Design Pattern

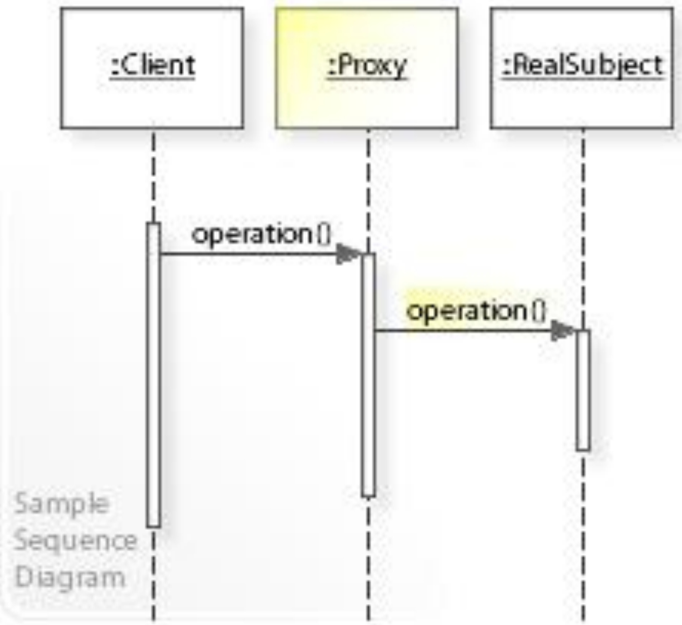


Dynamic Proxy Design Pattern

- A proxy, in its most general form, is a class functioning as an **interface to something else**. A proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide **additional logic**.
- In the proxy, **extra functionality can be provided**, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked.
- For the client, usage of a proxy object is similar to using the real object, because **both implement the same interface**.
- **Dynamic proxies** can be generated through reflection of bean methods, providing additional functionality

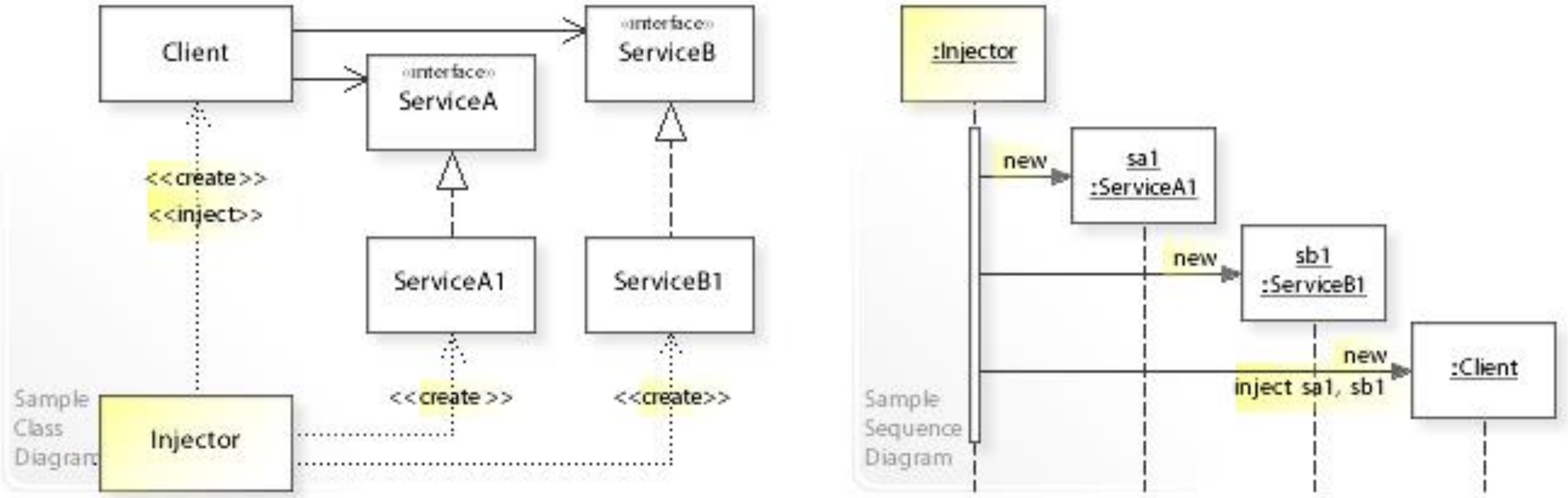
Dynamic Proxy Design Pattern





By Vanderjoe - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=62347717

Dependency Injection Design Pattern



By Vanderjoe - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=63061518

MVC Comes in Different Flavors

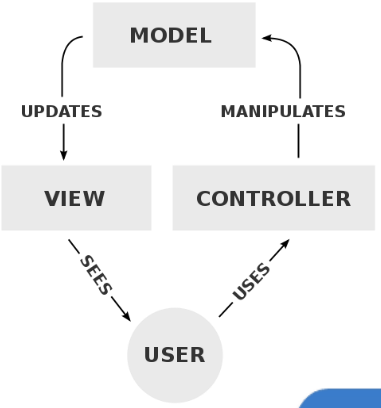
What is the difference between following patterns:

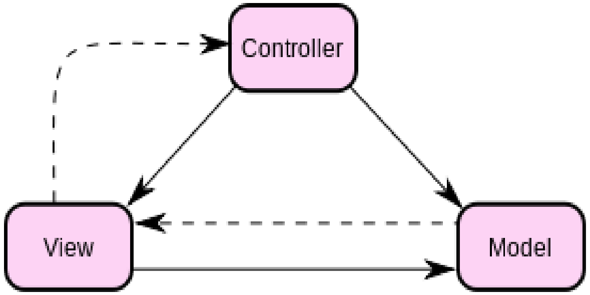
- Model-View-Controller (MVC)
- Model-View-ViewModel (MVVM)
- Model-View-Presenter (MVP)

http://csl.ensm-douai.fr/noury/uploads/20/ModelViewController.mp3

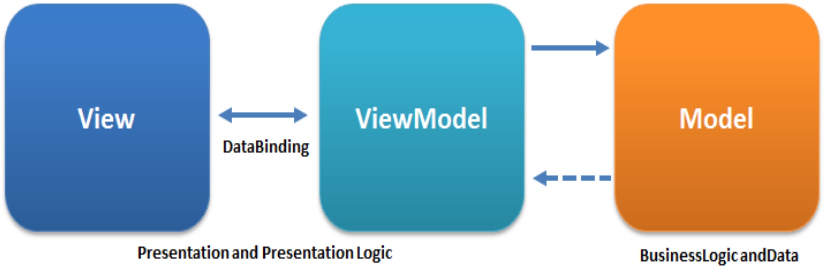
MVC Comes in Different Flavors - II

- MVC

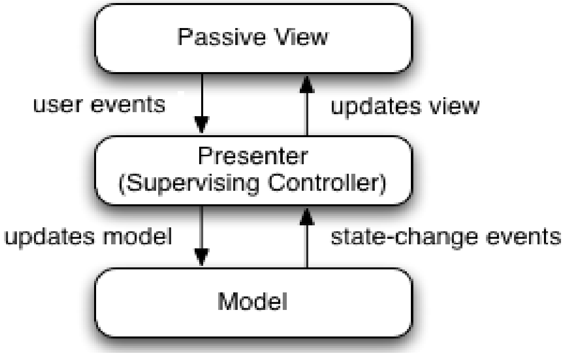




- MVVM

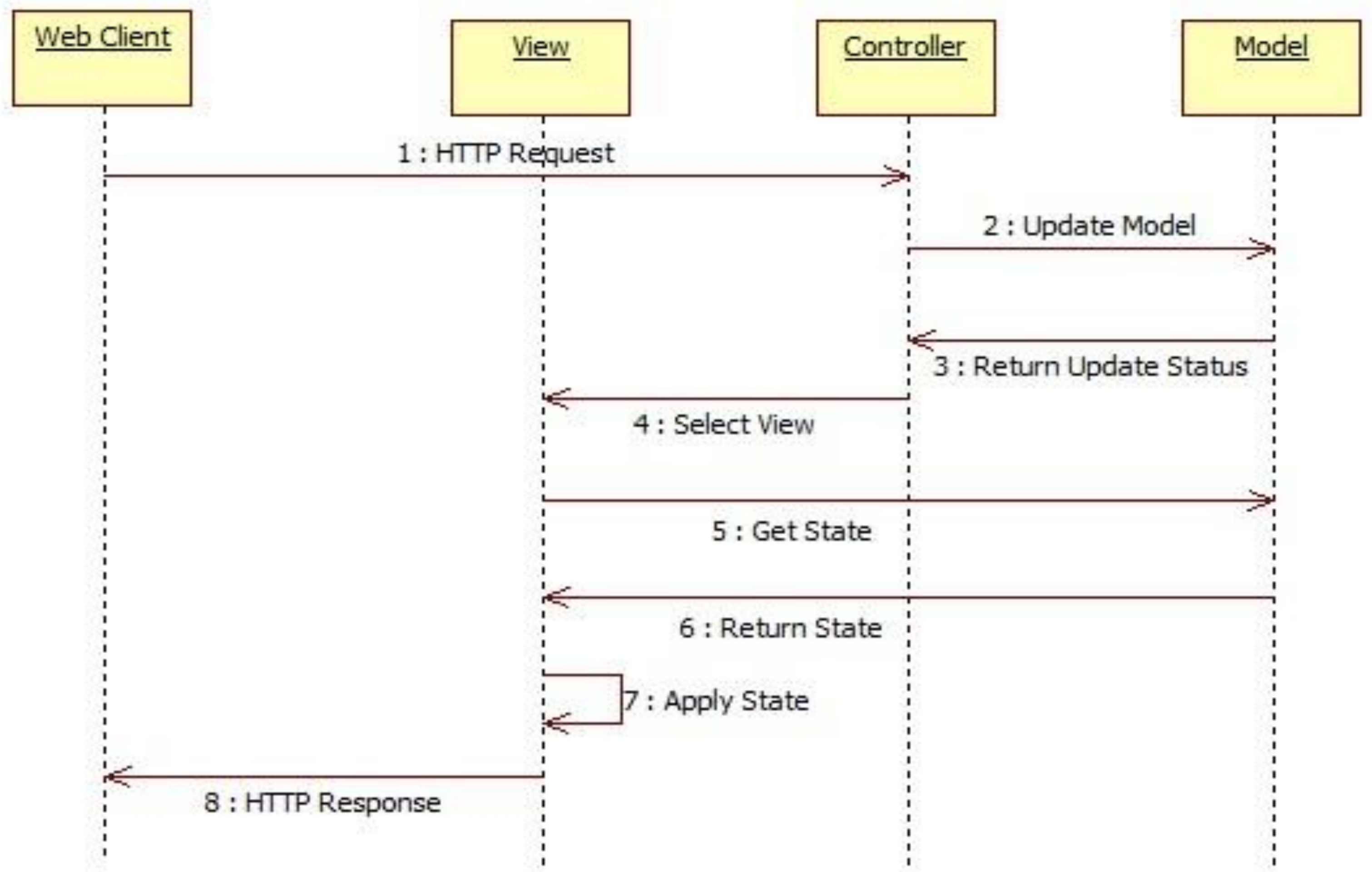


- MVP



Sources:https://en.wikipedia.org/wiki/Model_View_ViewModel#/media/File:MVVMPattern.png,
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter#/media/File:Model_View_Presenter_GUI_Design_Pattern.png
License: CC BY-SA 3.0, Authors:Ugaya40, Daniel.Cardenas

Web MVC Interactions Sequence Diagram



Domain Driven Design (DDD)

We need tools to cope with all that complexity inherent in robotics and IoT domains.

Simple solutions are needed – cope with problems through divide and concur on different levels of abstraction:

Domain Driven Design (DDD) – back to basics: domain objects, data and logic.

Described by Eric Evans in his book:
Domain Driven Design: Tackling Complexity in the Heart of Software, 2004

Domain Driven Design (DDD)

Main concepts:

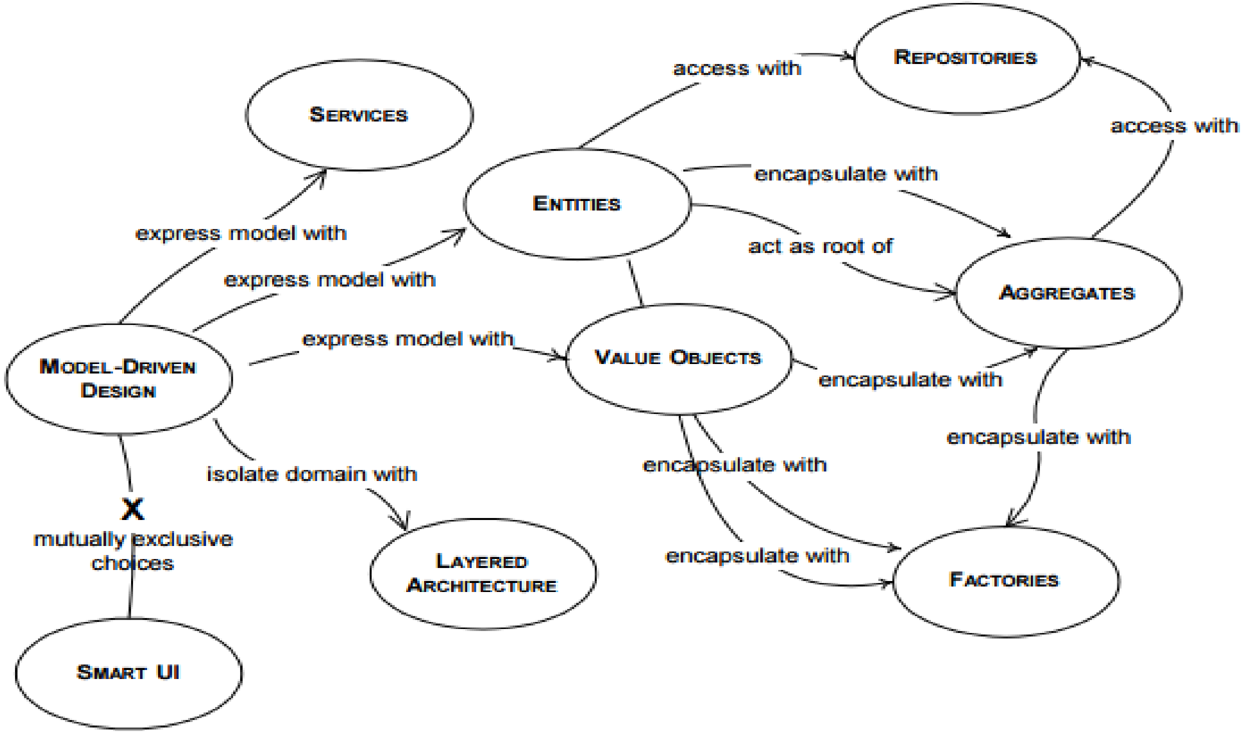
❖ Entities, value objects and modules

❖ Aggregates and Aggregate Roots [Haywood]:
value < entity < aggregate < module < BC

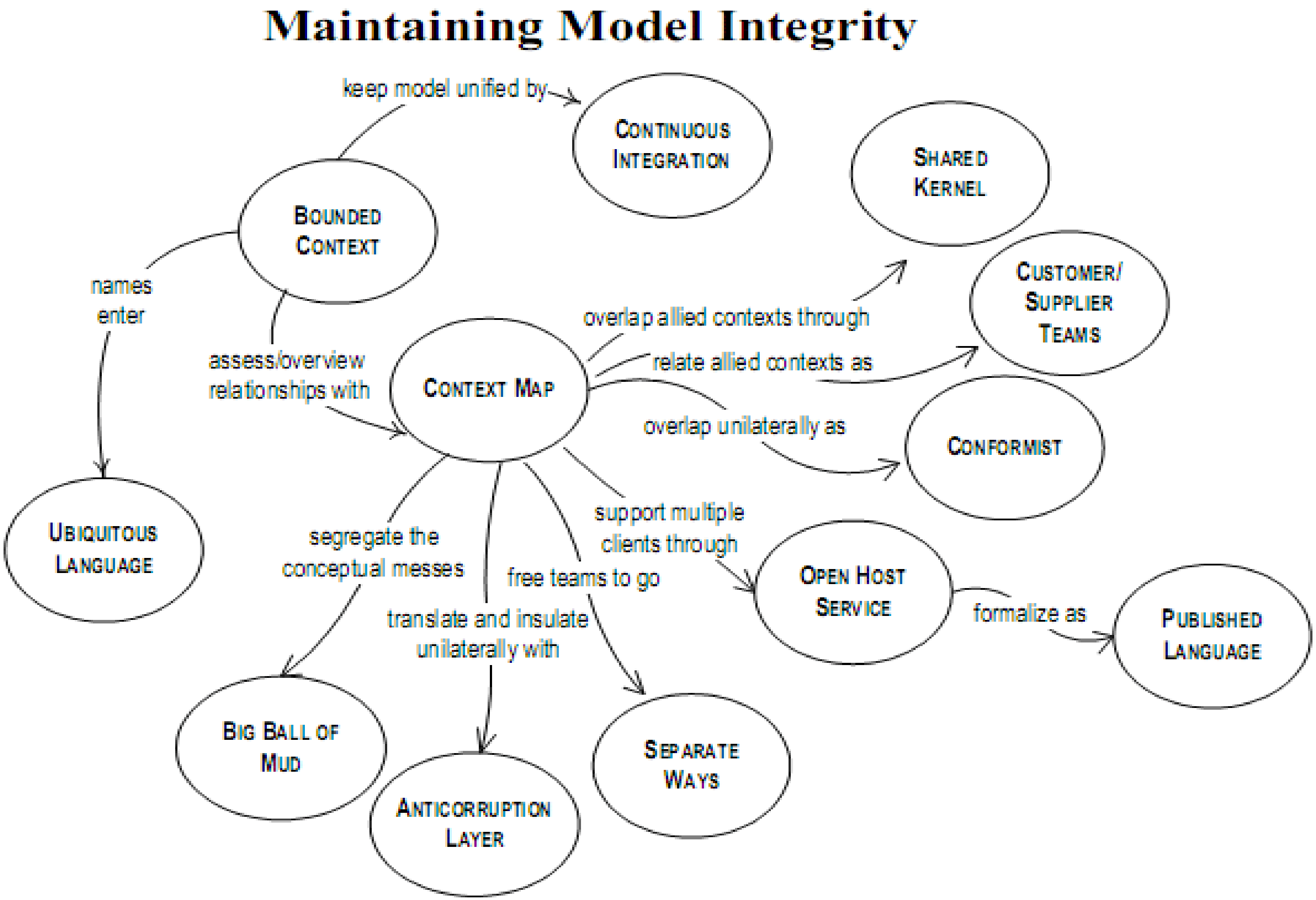
❖ Repositories, Factories and Services:
application services <-> **domain services**

❖ Separating interface from implementation

Domain Driven Design (DDD)



Domain Driven Design (DDD)



Domain Driven Design (DDD)

❖Ubiquitous language and Bounded Contexts

❖DDD Application Layers:

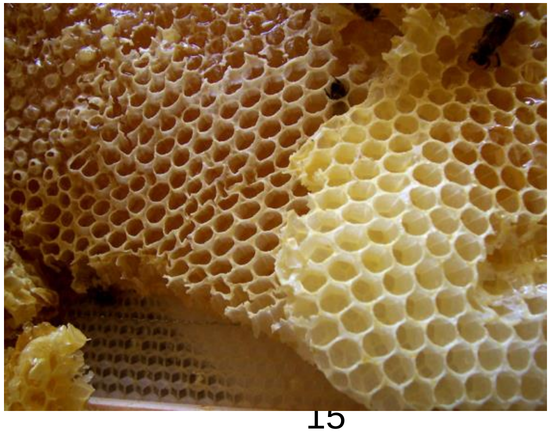
❖Infrastructure, Domain, Application, Presentation

❖ Hexagonal architecture :

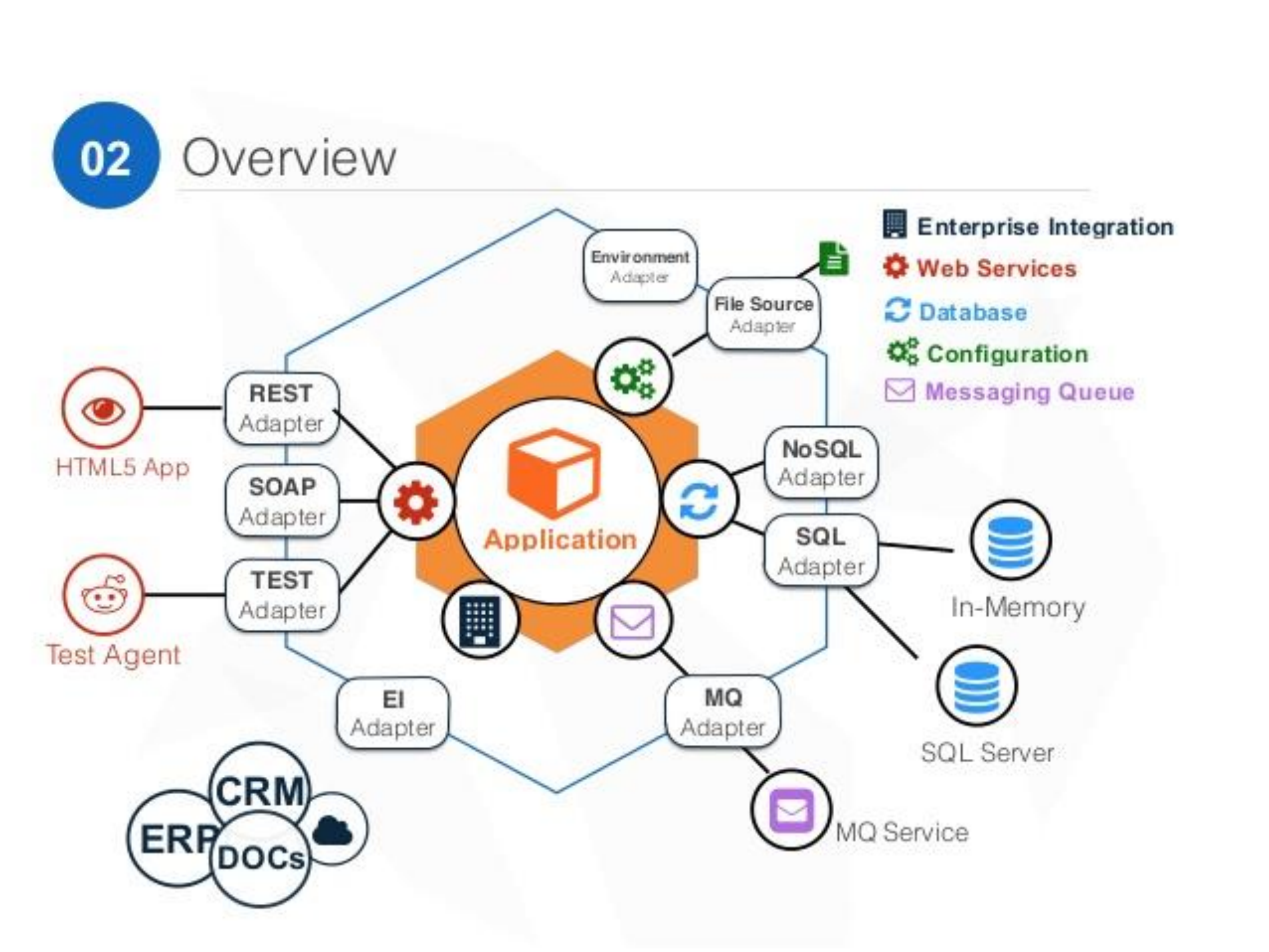
OUTSIDE <-> transformer <->

(application <-> domain)

[A. Cockburn]



Hexagonal Architecture



Source: https://www.slideshare.net/fabricioepa/hexagonal-architecture-for-java-applications

Hexagonal Architecture Principles

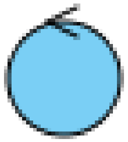
- ❖Allows an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.
- ❖As events arrive from the outside world at a port, a technology-specific adapter converts it into a procedure call or message and passes it to the application
- ❖Application sends messages through ports to adapters, which signal data to the receiver (human or automated)
- ❖The application has a semantically sound interaction with all the adapters, without actually knowing the nature of the things on the other side of the adapters

Source: http://alistair.cockburn.us/Hexagonal+architecture

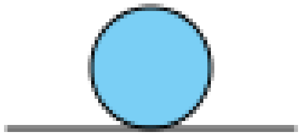
Analysis Classes Stereotypes

Analysis classes are used in the mapping and analysis of system architecture - they present rather different roles and responsibilities, than specific classes to be realized, and are independent of implementation technology:

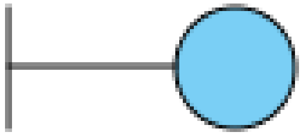
- <<controll>> - business logic
- <<entity>> - data
- <<boundary>> - user or system interface



Controlling Class



Class Unit

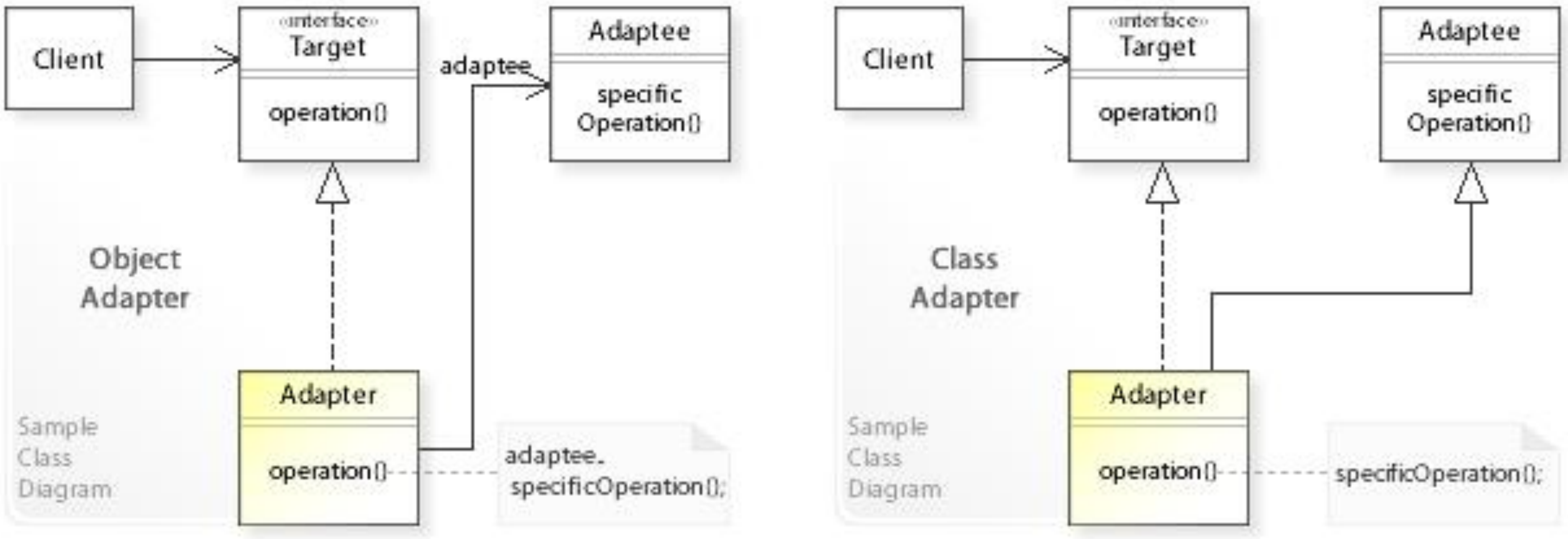


Border Class

Advantages of Using Interfaces

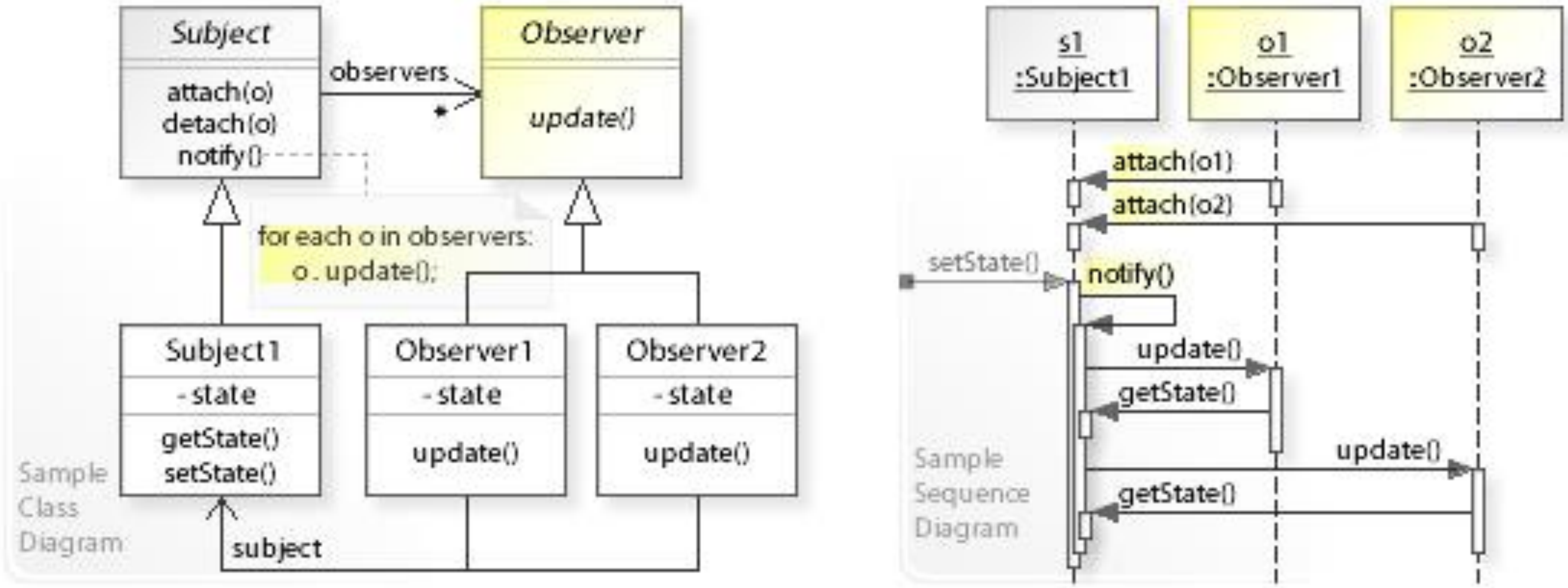
- Interfaces cleanly separate requirements type of the object from many possible implementations and make our code more universal and usable
- Reusable Design Pattern: Adapter – It allows to adapt existing realization interface that is required in our application
- Inheritance (expansion) of interfaces
- Reusable Design Pattern: Factory Method – creating reusable client code, isolated from the specifics of the particular server implementation

Adaptor Design Pattern



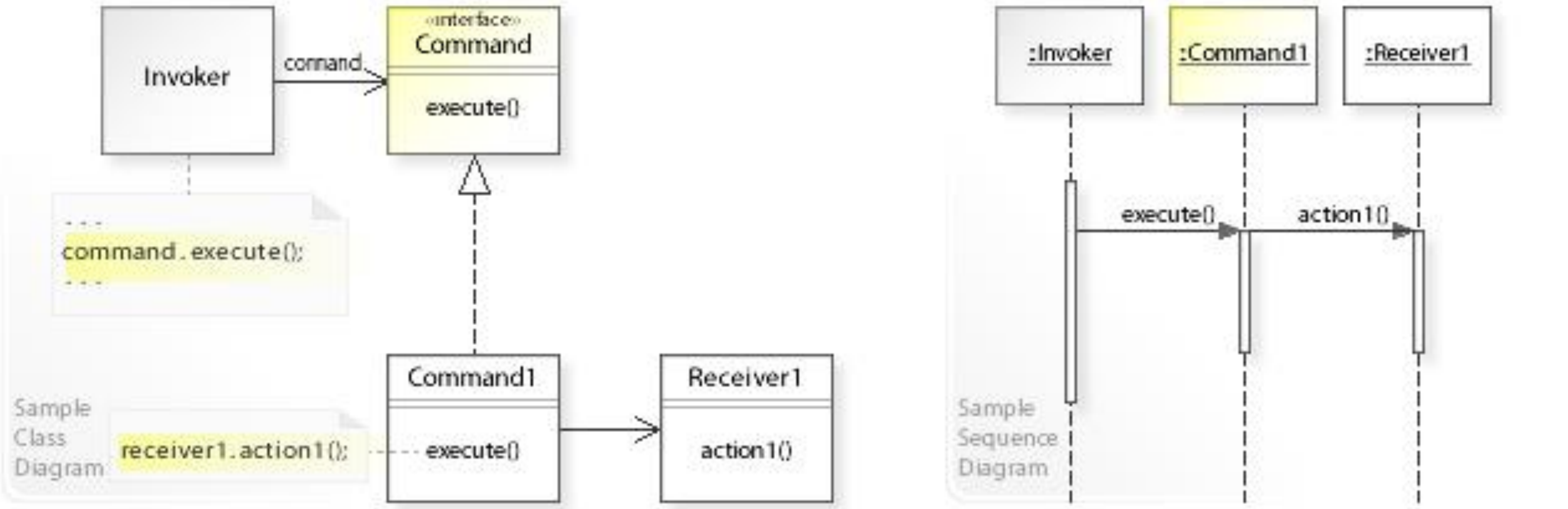
By Vanderjoe - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=63061518

Observer Design Pattern



By Vanderjoe - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=63061518

Command Design Pattern

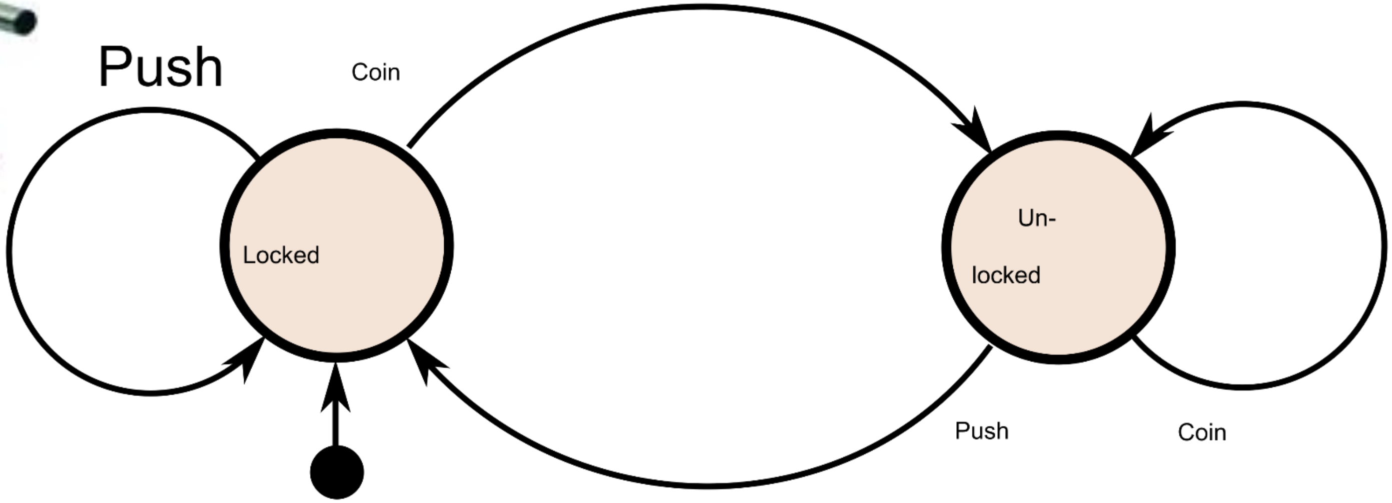


By Vanderjoe - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=63061518

State Design Pattern

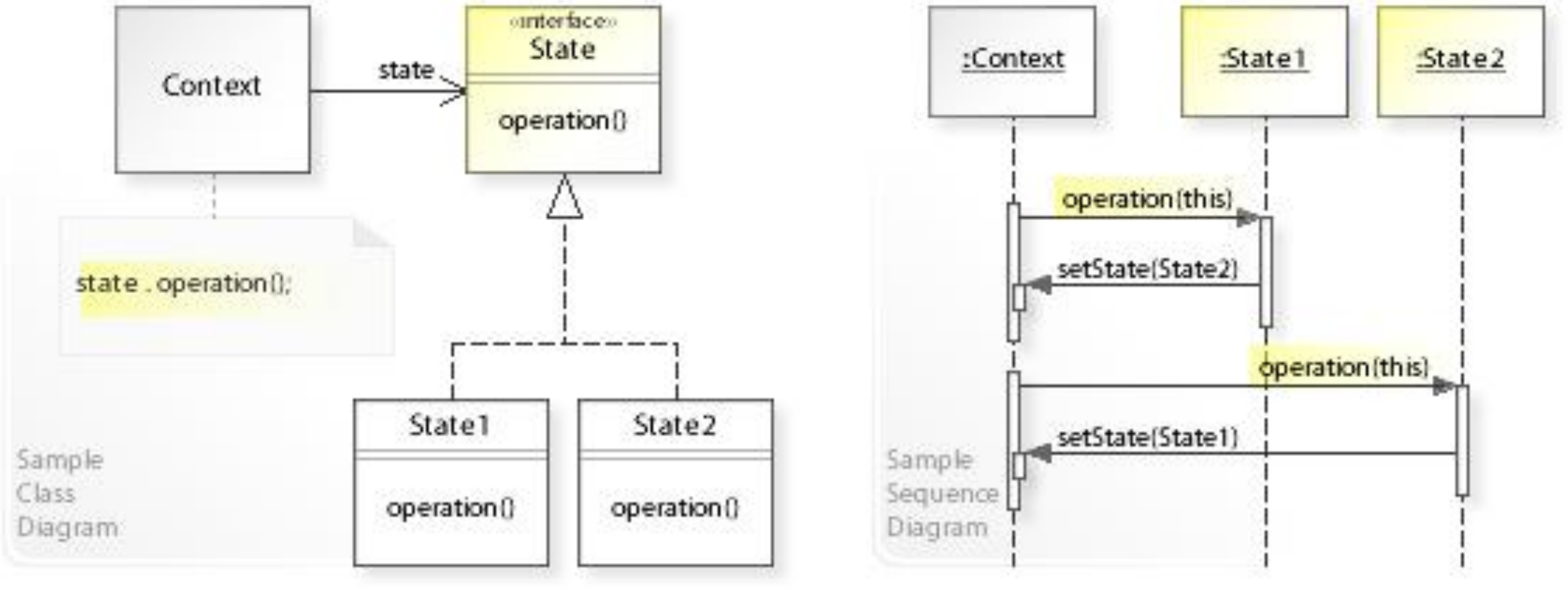
- An object should change its behavior when its internal state changes.
- State-specific behavior should be defined independently. That is, adding new states should not affect the behavior of existing states.
- Define separate (state) objects that encapsulate state-specific behavior for each state. That is, define an interface (state) for performing state-specific behavior, and define classes that implement the interface for each state.
- A class delegates state-specific behavior to its current state object instead of implementing state-specific behavior directly.

State Design Pattern



By Sebasgui - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=8930080

State Design Pattern



By Vanderjoe - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=63061518

Java I/O

- Input/Output from/to:

✓ Memory

✓ String

✓ Between different threads

✓ Files

✓ Console

✓ Network sockets

- Different data types – bytes / characters. Encoding.

- Common and extensible architecture of Java I/O system using Decorator design pattern.

Class **File** – Working with Files and Dirs

- Class ***File***
- Represents a file or a directory.
- Methods ***getName()*** and ***list()***
- Getting file information
- Creating, renaming and deleting directories.

Input and Output Streams

- Input streams – class ***InputStream*** and its inheritors
- Output streams – class ***OutputStream*** and its inheritors
- Decorator design pattern
- Decorators - class ***FilterInputStream*** and its inheritors, class ***FilterOutputStream*** and its inheritors

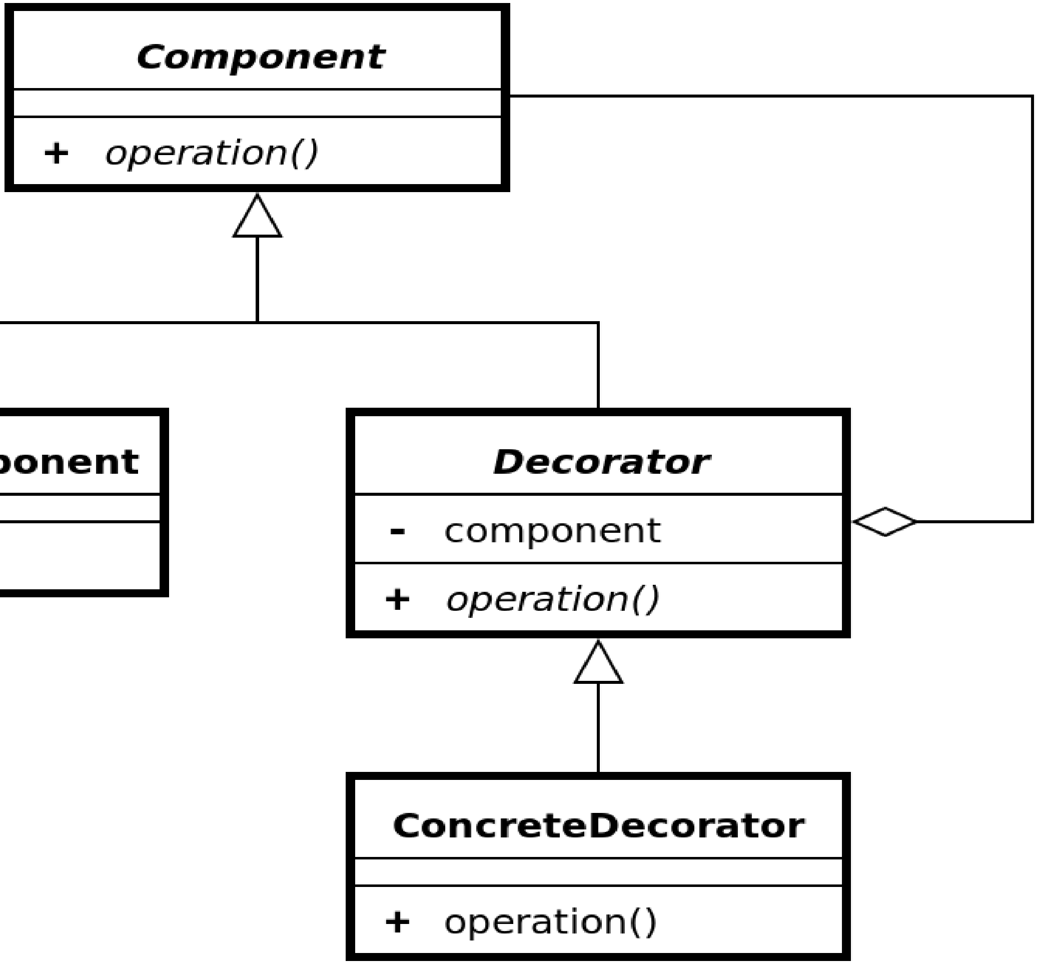
Input Streams: ***InputStream***

- **FileInputStream** – reads data from file
- **ByteArrayInputStream** – reads data from memory
- **StringBufferInputStream** – reads data from StringBuffer
- **ObjectInputStream** – de-serializes Objects and primitives
- **PipedInputStream** – receives data from another thread
- **SequenceInputStream** – combines several InputStreams
- **FilterInputStream** – decorates wrapped input streams with additional functionality

Output Streams: ***OutputStream***

- **FileOutputStream** – writes data to file
- **ByteArrayOutputStream** – writes data to memory buffer
- **ObjectOutputStream** – serializes objects and primitives
- **PipedOutputStream** – sends data to another thread
- **FilterOutputStream** – decorates wrapped InputStreams with additional functionality

Decorator Design Pattern



Input Stream Decorators

- **DataInputStream** – reads primitive types
- **BufferedInputStream** – buffers the input, allows reading lines instead of characters
- **DigestInputStream** – calculates content hash using algorithms such as: SHA-1, SHA-256, MD5
- **DeflaterInputStream** – data compression
- **InflaterInputStream** – data decompression
- **CheckedInputStream** – calculates checksum (Adler32, CRC32)
- **CipherInputStream** – decrips data (using Cipher)

Output Stream Decorators

- **PrintStream** – provides convenient methods for printing different data types, processes exceptions
- **DataOutputStream** – writes primitive data types
- **BufferedOutputStream** – output buffering
- **DigestOutputStream** – calculates content hash using algorithms such as: SHA-1, SHA-256, MD5
- **DeflaterOutputStream** – data compression
- **InflaterOutputStream** – data decompression
- **CheckedOutputStream** – checksum computation
- **CipherInputStream** – encrips data (using Cipher)

Reading Character Data: ***Reader***
Adaptor Class: ***InputStreamReader***

- **FileReader** – reads character data from file
- **CharArrayReader** - reads character data from memory
- **StringReader** – reads character data from String
- **PipedReader** – receives character data from a thread
- **FilterReader** – Reader decorator base class

Writing Character Data : ***Writer***
Adaptor Class: ***OutputStreamWriter***

- **FileWriter** – writes character data to file

- **CharArrayWriter** - writes character data to array

- **StringWriter** – writes character data to StringBuffer

- **PipedWriter** – sends character data to another thread

- **FilterWriter** – base class for Writer decorators

- **PrintWriter** – formatted output in string format, handles all exceptions

Reader / Writer Decorators

- **BufferedReader** – character input buffering
- **PushbackReader** – allows characters to be read without consuming
- **BufferedWriter** – character output buffering
- **StreamTokenizer** – allows parsing of character input (from Reader) token by token

Direct Access Files

- Class ***RandomAccessFile***.
- Access modes
- Method ***seek()***
- Usage examples.

- Standard I/O to/from console. Redirecting.

Novelties in Java 7 - JSR 203: NIO.2 (1)

- New NIO packages: java.nio.file, java.nio.file.attribute
- FileSystem – allows a unified aceess to different file systems using URI or the method FileSystems.getDefault(). A factory for file system object creation. Methods: getPath(), getPathMatcher(), getFileStores(), newWatchService(), getUserPrincipalLookupService().
- FileStore – models a drive, partition or a root directory. Can be accessed using FileSystem.getFileStores()

Novelties in Java 7 - JSR 203: NIO.2 (2)

- Path – represents a file or directory path in the file system. Has a hierarchical structure – a sequence of directories separated using an OS specific separator ('/' или '\'). Provides methods for composing, decomposing, comparing, normalizing, transforming relative and absolute paths, watching for file and directory changes, conversion to/from File objects (java.io.File.toPath() и Path.toFile()).
- Files – utility class providing static methods for manipulation (creation, deletion, renaming, attributes change, content access, automatic MIME type inference, etc.) of files, directories, symbolic links, etc.

Exercise: Walking + Filtering File Tree Using Streams

Using the java.nio.file.Files.walk(Path start, FileVisitOption... options) method, implement following functionality:

1. Walk the ./src directory of the project and print names of all files with .java extension recursively in all subdirectories.
2. Calculate the total number and size statistics (min / max / average size of java files – in terms of **number of characters** and **lines of code**, and the sum of sizes – tolal lines of code) of all java files in the project. The empty lines are **not** counted in the statistics.

Compression: GZIP, ZIP. JAR Files

- File compression – gzip, zip. Check Sum.
- Application deployment using .jar archives. JAR file manifest.
- **jar** [options] archive [manifest] files

- **c** – creates new archive
- **x / x файл** – extracts specific/all files from an archive
- **t** – prints archive content table
- **f** – necessary to specify the file we read/write from/to
- **m** – if we provide a manifest file
- **M** – do not create manifest file automatically
- **0** – without compression
- **v** – verbose output

Object Serialization

- Interface ***Serializable* –** all fields are serialized except those marked as ***transient***
- Interface ***Externalizable* –** we serialize all fields explicitely
- Methods ***readObject()*** and ***writeObject()* – *Serializable*** + customization where necessary
- Examples

Resources

- New I/O във Wikipedia: http://en.wikipedia.org/wiki/New_I/O
- Уроци за новостите в JSR 203: NIO.2
  http://download.oracle.com/javase/tutorial/essential/io/fileio.html
- Joshua Bloch: Automatic Resource Management (V.2) – https://docs.google.com/View?id=ddv8ts74_3fs7483dp

Thank’s for Your Attention!

Trayan Iliev

CEO of IPT – Intellectual Products

& Technologies

http://iproduct.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD

https://plus.google.com/+IproductOrg